**Лекція 17. Механізми та методи тестування. Тестові приклади.**

**Тестування ПЗ** (Software testing) – перевірка відповідності між реальною і очікуваною поведінкою програми.

Тестування – це процес дослідження ПЗ з метою виявлення помилок і перевірки якості.

Тестування – це одна з технік контролю якості, що включає в себе активності з планування робіт (Test Management), проектування тестів (Test Design), виконання тестування (Test Execution) та аналізу отриманих результатів (Test Analysis).

Тестування так само можна описати як процес верифікації та валідації того чи іншого програмного продукту, щоб дізнатися на скільки точно він задовольняє всім встановленим вимогам.

**Верифікація** (Verification – узгодження) – це процес оцінки системи або її компонентів з метою визначення чи задовольняють результати поточного етапу розробки умовам, сформованим на початку цього етапу (чи виконуються наші цілі, терміни, завдання, по розробці проекту, визначені на початку поточної фази.)

**Валідація** (Validation – затвердження) – це визначення відповідності ПЗ очікуванням і потребам користувача, вимогам до системи.

Незважаючи на схожість, терміни «тестування», «верифікація» і «валідація» означають різні рівні перевірки коректності роботи програмної системи.



**Рисунок 1 –** Тестування, верифікація і валідація

Якщо ще раз подивитися на ці три процеси з точки зору питання, на яке вони дають відповідь, то тестування відповідає на питання «*Як це зроблено?*» або «*Чи відповідає поведінка розробленої програми вимогам?*», верифікація – «*Що зроблено?*» або «*Чи відповідає розроблена система вимогам?*», а валідація – «*Чи зроблено те, що потрібно?*» або «*Чи відповідає розроблена система очікуванням замовника?*».

**Цілі і завдання процесу тестування**

Основною метою процесу тестування – є доказ того, що результат розробки відповідає пред’явленим до нього вимогам.

Основне завдання тестування ПЗ: отримання інформації про статус готовності заявленої функціональності системи або програми.

**Необхідні і достатні умови для проведення тестування**

*Необхідними умовами істинності* твердження А називаються умови, без дотримання яких А не може бути істинним.

*Достатніми називаються такі умови*, за наявності (виконанні, дотриманні) яких твердження А є істинним.

Необхідні умови:

• Наявність об’єкта тестування, доступного для проведення випробувань.

• Наявність виконавця (ів) (людина або машина, або комбінація людина + машина)

Достатні умови:

• Наявність об’єкта тестування, доступного для проведення випробувань

• Наявність виконавця (ів) (людина або машина, або комбінація людина + машина)

• Наявність плану тестування

• Наявність тест кейсів / тестів

• Наявність звіту, що підтверджує виконання завдань і досягнення цілей, з тестування об’єкта

**План Тестування** (Test Plan) – це документ, що описує весь обсяг робіт з тестування, починаючи з опису об’єкта, стратегії, розкладу, критеріїв початку і закінчення тестування, до необхідного в процесі роботи обладнання, спеціальних знань, а також оцінки ризиків з варіантами їх дозволу .

**Тестовий випадок** (Test Case) – це артефакт, що описує сукупність кроків, конкретних умов і параметрів, необхідних для перевірки реалізації функції, що тестується або її частини.

Основні атрибути Test Case:

1) ID (номер),

2) Name (ім’я),

3) Preconditions (умови і параметри),

4) Steps (кроки до відтворення),

5) Expected result (очікуваний результат),

6) Postconditions (постумови)

**Тест дизайн** (Test Design) – це етап процесу тестування ПЗ, на якому проектуються і створюються тестові випадки (тест кейси), відповідно з визначеними раніше критеріями якості і цілями тестування.

**Баг / Дефект Репорт** (Bug Report) – це документ, що описує ситуацію або послідовність дій, що призвела до некоректної роботи об’єкта тестування, із зазначенням причин і очікуваного результату.

**Тестове Покриття** (Test Coverage) – це одна з метрик оцінки якості тестування, що представляє із себе щільність покриття тестами.

**Специфікація Тест Кейсів** (Test Case Specification) – це рівень деталізації опису тестових кроків і необхідного результату, при якому забезпечується розумне співвідношення часу проходження до тестового покриття

**Час Проходження Тест Кейса** (Test Case Pass Time) – це час від початку проходження кроків тест кейса до отримання результату тесту.

**Тестові дані**

Усяке ПЗ обробляє певні дані, які можна визначити, як відомості про деяку подію, факт тощо, що представлені у формальному вигляді і підлягають подальшому аналізу. Аналіз і перетворення даних і являють собою основні завдання ПЗ (важливо пам’ятати різницю між даними та інформацією, яку можна отримати на підставі даних). Таким чином, для проведення тестування ПЗ необхідні дані, які б симулювали реальний потік, оброблюваний програмою.

Існує два типи тестових даних: *реальні та синтезовані*. Реальні дані можна отримати двома способами: перший полягає в тому, щоб тестувати ПЗ в реальних умовах (що практично неможливо); другий – отримувати реальні дані у кінцевого користувача (що можливо досить рідко) або з існуючої системи (у цьому випадку можуть виникнути проблеми з конфіденційністю). Тому, найчастіше для тестування використовують синтезовані дані.

Основною умовою синтезу даних є їх правдоподібність. Синтезувати дані можна на підставі зразка (невеликої кількості реальних даних), статистичної інформації про те, які дані є типовими для конкретного тестового випадку, а також іншого набору умов, що обмежують спектр допустимих варіацій вхідних даних. На базі наявної інформації будується генератор тестових даних, який формує необхідний для тестування потік.

**Тестова ситуація**

Тестова ситуація – це базисне поняття тестування. *Під тестовою ситуацією розуміється деякий певний стан системи, що тестується, якому відповідає строго певний набір її параметрів.* Тестову ситуацію можна визначити також як сукупність зовнішніх і внутрішніх факторів, що дозволяють розрізнити умови виконання програми. Якщо один з параметрів змінюється – зміниться і тестова ситуація. Виходячи з цього, повний набір всіх можливих тестових ситуацій визначає тестове покриття.

**Типи тестових прикладів**

Розглянемо різні класи тестових прикладів, спрямованих на виявлення різних дефектів в роботі програмної системи.

**Допустимі дані**

Найчастіше дефекти в програмних системах проявляються при обробці нестандартних даних, що не передбачені вимогами – при введенні невірних символів, порожніх рядків, дуже великій швидкості введення інформації. Перед пошуком таких дефектів необхідно упевнитися в тому, що програма коректно обробляє вірні дані, передбачені специфікацією, тобто перевірити роботу основних алгоритмів.

Приклад 1. Для функції обчислення контрольної суми допустимими вхідними даними буде довільний запис, що містить дані у всіх полях, крім поля контрольної суми CRC. Сценарієм буде виклик функції розрахунку Set\_CRC, яка обчислює контрольну суму передану через структуру з усіма потрібними для розрахунку даними, а очікуваним вихідним значенням – коректне значення поля CRC, що розраховане за алгоритмом.

Зазвичай для перевірки допустимих даних достатньо одного тестового прикладу. Але функціональні вимоги можуть визначати різні групи допустимих даних, які можуть об’єднуватися в класи еквівалентності. У цьому випадку необхідно визначати як мінімум один тестовий приклад для одного класу еквівалентності.

**Класи еквівалентності (Equivalence class):**

Підхід полягає в наступному: вхідні / вихідні дані розбиваються на класи еквівалентності, за принципом, що програма веде себе однаково з кожним представником окремого класу. Таким чином, немає необхідності тестувати всі можливі вхідні дані, необхідно перевірити по окремо взятому представнику класу.

Клас еквівалентності – це набір значень змінної, який вважається еквівалентним.

Тестові сценарії еквівалентні, якщо:

* Вони тестують одне і те ж;
* Якщо один з них знаходить помилку, то й інші виявлять її;
* Якщо один з них не знаходить помилку, то й інші не виявлять її.

**Еквівалентне розбиття:** Розробка тестів методом чорного ящика, в якому тестові сценарії створюються для перевірки елементів еквівалентної області. Як правило, тестові сценарії розробляються для покриття кожної області як мінімум один раз.

[](http://qalearning.com.ua/wp-content/uploads/2015/01/2.png)

Приклад 2:

Припустимо, ми тестуємо Інтернет-магазин, який продає олівці. У замовленні необхідно вказати кількість олівців (максимум для замовлення – 1000 штук). Залежно від замовленої кількості олівців змінюється вартість:

1 – 100 – 10 грн. за олівець;

101 – 200 – 9 грн. за олівець;

201 – 300 – 8 грн. за олівець і т.д.

З кожною новою сотнею, ціна зменшується на гривню.

Якщо тестувати «в лоб», то, щоб перевірити всі можливі варіанти обробки замовленої кількості олівців, потрібно написати дуже багато тестів (згадуємо, що можна замовити аж 1000 штук), а потім ще все це і протестувати. Спробуємо застосувати розбиття на класи еквівалентності. Очевидно, що наші вхідні дані можна розділити на наступні класи еквівалентності:

Невалідне значення:> 1000 штук;

Невалідне значення: <= 0;

Валідне значення: від 1 до 100;

Валідне значення: від 101 до 200;

Валідне значення: від 201 до 300;

Валідне значення: від 301 до 400;

Валідне значення: від 401 до 500;

Валідне значення: від 501 до 600;

Валідне значення: від 601 до 700;

Валідне значення: від 701 до 800;

Валідне значення: від 801 до 900;

Валідне значення: від 901 до 1000.

На основі цих класів ми і складемо тестові сценарії. Отже, якщо взяти по одному представнику з кожного класу, то отримуємо 12 тестів.

**Граничні дані**

Окремий вид допустимих даних, передача яких в систему може розкрити дефект, – граничні дані, тобто, наприклад, числа, значення яких є граничними для їх типу, рядки граничної або нульової довжини тощо. Зазвичай за допомогою тестування граничних умов виявляються проблеми з арифметичним порівнянням чисел або з ітераторами циклів. Для тестування функції Set\_CRC з прикладу 1 на граничних умовах можна визначити два тестових приклади з мінімальними і максимальними значеннями полів у записі.

**Відсутність даних**

Дефекти можуть проявитися і у випадку, якщо системі не передається ніяких даних або передаються дані нульового розміру. Для тестування функції Set\_CRC при відсутності даних можна викликати її, передавши як параметр неініціалізованої структури. Однак такий тест не є точним прикладом відсутності даних, скоріше це приклад випадкових даних (можливо – невірних).

**Повторне введення даних**

У разі повторної передачі на вхід системи одних і тих же самих даних можуть виходити відмінності у вихідних даних, які не передбачені у вимогах. Як правило, *дефекти такого типу проявляються в результаті того, що система не встановлює внутрішні змінні в початковий стан або в результаті помилок округлення*.

**Невірні дані**

При перевірці поведінки системи необхідно не забувати перевіряти систему при передачі їй даних, що не передбачені вимогами – занадто довгих або занадто коротких рядків, невірних символів, чисел за межами діапазону тощо. Невірні дані, як і допустимі, також можна розділяти на різні класи еквівалентності. Прикладом невірних даних для функції Set\_CRC може служити запис з іншою структурою, переданій у функцію через приведення типів. Якщо розрахунок контрольної суми використовує імена полів запису, то контрольна сума може виявитися обчисленою невірно або може відбутися перезапис областей пам’яті, що не призначені для зберігання даних.

**Реініціалізація системи**

Механізми повторної ініціалізації системи під час її роботи також можуть містити дефекти. У першу чергу ці дефекти можуть проявлятися в тому, що не всі внутрішні дані системи після реініціалізаціі повернуться у початковий стан. У результаті може відбутися збій у роботі системи.

С**тійкість системи**

Під стійкістю системи розуміють її здатність витримувати нештатне навантаження, яке явно не передбачене вимогами. Наприклад, чи збереже система працездатність після 10 тисяч викликів.

Аналогічний аналіз може бути зроблений шляхом перегляду тексту програми (якщо він доступний при тестуванні) на підставі відсутності «історії» (збережених даних) в реалізації програми, тобто даних, значення яких може змінюватися в залежності від кількості запусків програми. Таким чином, у ряді випадків тестування може бути замінено аналізом програмного коду.

**Позаштатні стани середовища виконання**

Позаштатні стани середовища виконання (наприклад, вичерпання пам’яті, дискового простору або тривала нестача процесорного часу) можуть ускладнювати роботу системи або робити її неможливою. Основне завдання системи в такій ситуації – коректно завершити або призупинити свою роботу.

Тестовим прикладом, що створює нештатний стан середовища для функції Set\_CRC, може служити виділення всієї вільної пам’яті перед викликом функції. Якщо Set\_CRC використовує динамічну пам’ять, то в ній повинні бути перевірки на можливість виділити пам’ять, в іншому випадку виконання функції викличе її аварійне завершення:

*record\_type test\_value9;*

*int i;*

*int \*heap;*

*heap = malloc(\_MAXMEM);*

*test\_value9.A = false;*

*for (i=0;i<20;i++)*

*test\_value9.B[i] = i;*

*for (i=0;i<5;i++)*

*test\_value9.C[i] = i+5;*

*test\_value9.D[0] = i+8;*

*test\_value9.CRC = 0;*

*Set\_CRC(test\_value9);*

*free(heap);*

*printf(“%d\n”, test\_value9.CRC);*

**Граничні умови**

У тестових прикладах, що відповідають тест-вимогам, зазвичай використовуються вхідні значення, що знаходяться завідомо всередині допустимого діапазону. Один із способів перевірки стійкості системи на значеннях близьких до граничних – створювати для кожного входу як мінімум три тестових приклади.

* значення всередині діапазону;
* мінімальне значення;
* максимальне значення.

Для ще більшої впевненості в працездатності системи використовують п’ять тестових прикладів:

* значення всередині діапазону;
* мінімальне значення;
* мінімальне значення + 1;
* максимальне значення;
* максимальне значення – 1.

Такий спосіб перевірки називається перевіркою на граничних значеннях. Така перевірка дозволяє виявляти проблеми, що пов’язані з виходом за межі діапазону.

Наприклад, якщо в функцію,

*char sum(char a, char b)*

*{*

*return a+b;*

*}*

яка обчислює суму чисел *a* і *b* будуть передані значення 255 і 255, то в разі відсутності спеціальної обробки ситуації переповнення сума буде обчислена невірно.

Інша область, при тестуванні якої корисно користуватися перевіркою на граничних значеннях, – індекси масивів. Наприклад, функція,

*void abs\_array(char array[], char size)*

*{*

*for (int i=1;i<=size;i++)*

*{*

*array[i] = abs(array[i]);*

*}*

*return;*

*}*

яка замінює значення на значення по модулю у кожного елементу переданого їй масиву, містить помилку в циклі *for*, яка може бути легко виявлена при передачі у функцію масиву одиничного розміру.

**Вимоги до критерію тестування**

Він повинен бути:

* **достатнім**, тобто, коли деяка кінцева множина тестів достатня для тестування конкретно обраної програми;
* **повним**, тобто у разі помилки повинен існувати тест з множини тестів, що задовольняють критерію, який виявляє помилку;
* **надійним**, тобто будь – які дві множини тестів, що задовольняють йому, одночасно повинні розкривати або не розкривати помилки програми;
* таким, що **легко перевіряється**, наприклад, обчислюваним на тестах.

Для нетривіальних класів програм у загальному випадку не існує повного і надійного критерію, що залежить від програм або специфікацій. Тому прагнуть до ідеального загального через реальні часткові критерії, які можна розбити на такі класи.

1. Структурні критерії, що використовують інформацію про структуру програми (критерії так званого «білого ящика» ).

2. Функціональні критерії, що формулюються в описі вимог до програмного виробу (критерії так званого «чорного ящика» )

3. Критерії стохастичного тестування, що формулюються в термінах перевірки наявності заданих властивостей у тестованому додатку, засобами перевірки деякої статистичної гіпотези.

4. Мутаційні критерії, що орієнтовані на перевірку властивостей програмного виробу на основі підходу Монте–Карло.

**Особливості застосування структурних і функціональних критеріїв**

***Структурні критерії (клас I).***

Структурні критерії використовують модель програми у вигляді «білого ящика», що припускає знання початкового тексту програми або специфікації програми у вигляді потокового графа управління. Структурна інформація зрозуміла і доступна розробникам підсистем і модулів додатку, тому даний клас критеріїв часто використовується на етапах модульного і інтеграційного тестування (Unit testing, Integration testing).

Структурні критерії базуються на основних елементах управляючого графа програми, операторах, гілках і шляхах.

Умовою критерію **тестування команд** є те, що набір тестів у сукупності повинен забезпечити проходження кожної команди не менше одного разу. Це слабкий критерій, він, як правило, використовується у великих програмних системах, де інші критерії застосувати неможливо.

Умовою критерію **тестування гілок** є те, що набір тестів у сукупності повинен забезпечити проходження кожної гілки не менше одного разу. Це досить сильний і при цьому економічний критерій, оскільки множина гілок у тестованому додатку і не така вже велика. Даний критерій часто використовується в системах автоматизації тестування.

Умовою критерію **тестування шляхів** є те, що набір тестів в сукупності повинен забезпечити проходження кожного шляху не менше одного разу. Якщо програма містить цикл (особливо з неявно заданим числом ітерацій), то число ітерацій обмежується константою (часто – 2, або числом класів вихідних шляхів).

**Функціональні критерії (клас II).**

**Функціональний критерій** – найважливіший для програмної індустрії критерій тестування. Він забезпечує, перш за все, контроль ступеня виконання вимог замовника в програмному продукті. Оскільки вимоги формулюються до продукту в цілому, вони відображають взаємодію тестованого додатка з оточенням. При функціональному тестуванні переважно використовується модель «чорного ящика». Проблема функціонального тестування – це, перш за все, трудомісткість, оскільки документи, які фіксують вимоги до програмного виробу (Software requirement specification, Functional specification тощо), як правило, достатньо об’ємні, проте, відповідна перевірка повинна бути всеосяжною.

Основні види функціональних критеріїв.

**Тестування пунктів специфікації –** набір тестів у сукупності повинен забезпечити перевірку кожного тестованого пункту не менше одного разу.

Специфікація вимог може містити сотні і тисячі пунктів вимог до програмного продукту і кожна з цих вимог при тестуванні повинна бути перевірена відповідно до критерію не менше ніж одним тестом

**Тестування класів вхідних даних** – набір тестів у сукупності повинен забезпечити перевірку представника кожного класу вхідних даних не менше одного разу.

**Тестування пунктів специфікації –** набір тестів у сукупності повинен забезпечити перевірку кожного тестованого пункту не менше одного разу.

Специфікація вимог може містити сотні і тисячі пунктів вимог до програмного продукту і кожна з цих вимог при тестуванні повинна бути перевірена відповідно до критерію не менше ніж одним тестом

**Тестування класів вхідних даних** – набір тестів у сукупності повинен забезпечити перевірку представника кожного класу вхідних даних не менше одного разу.

Граматика повинна бути достатньо простою, щоб трудомісткість розробки відповідного набору тестів була реальною (вписувалася в терміни і штат фахівців, виділених для реалізації фази тестування).

**Тестування класів вихідних даних** – набір тестів в сукупності повинен забезпечити перевірку представника кожного вихідного класу, за умови, що вихідні результати наперед класифікуються, причому окремі класи результатів враховують, зокрема, обмеження на ресурси або на якийсь час (time out).

**Тестування функцій** – набір тестів у сукупності повинен забезпечити перевірку кожної дії, що реалізовується тестованим модулем, не менше одного разу.

Функціональний критерій дуже популярний на практиці, але не забезпечує покриття частини функціональності тестованого компонента, яка пов’язана із структурними і поведінковими властивостями, опис яких не зосереджений в окремих функціях (тобто опис розосереджений по компоненту).

Критерій тестування функцій об’єднує частково особливості структурних і функціональних критеріїв. Він базується на моделі «напівпрозорого ящика», де явно вказані не тільки входи і виходи тестованого компонента, але також склад і структура використовуваних методів (функцій, процедур) і класів.

Комбіновані критерії для програм і специфікацій – набір тестів в сукупності повинен забезпечити перевірку всіх комбінацій несуперечливих умов програм і специфікацій не менше одного разу.

При цьому всі комбінації несуперечливих умов треба підтвердити, а умови суперечностей слід виявити і ліквідовувати.

**Застосування методів стохастичного тестування**

***Стохастичні критерії (клас III).***

Стохастичне тестування застосовується при тестуванні складних програмних комплексів, коли набір детермінованих тестів (X, Y) має велику потужність. У випадках, коли подібний набір неможливо розробити і виконати на фазі тестування, можна застосувати таку методику.

Розробити програми-імітатори випадкових послідовностей вхідних сигналів {*x*}.

Обчислити незалежним способом значення {*y*} для відповідних вхідних сигналів {*x*} і отримати тестовий набір (X, Y).

Протестувати додаток на тестовому наборі (X, Y), використовуючи два способи контролю результатів:

**Детермінований контроль** – перевірка відповідності обчисленого значення yвих ∈{*y*} значенню *у*, отриманому в результаті прогону тесту на наборі {*x*} – випадковій послідовності вхідних сигналів, що згенерована імітатором.

**Стохастичний контроль** – перевірка відповідності множині значень {yвих}, отриманих у результаті прогону тестів на наборі вхідних значень {*x*}, наперед відомому розподілі результатів F(Y).

У цьому випадку множина Y невідома (її обчислення неможливе), але відомий закон розподілу даної множини.

***Критерії стохастичного тестування.***

**Статистичні методи закінчення тестування** – стохастичні методи ухвалення рішень про збіг гіпотез про розподіл випадкових величин. До них належить широко відомий критерій Стьюдента (**St**), метод Хі–квадрат (χ2) та ін.

**Метод оцінки швидкості виявлення помилок** – заснований на моделі швидкості виявлення помилок, згідно з якою тестування припиняється, якщо оцінений інтервал часу між поточною помилкою і наступною дуже великий для фази тестування додатку

**Мутаційний критерій і техніка роботи з ним**

***Мутаційний критерій (клас IV).***

Професійні програмісти пишуть відразу майже правильні програми, що відрізняються від правильних дрібними помилками, наприклад – перестановка місцями максимальних значень індексів в описі масивів, помилки в знаках арифметичних операцій, заниження або завищення межі циклу на 1 тощо. Розглянемо підхід, що дозволяє на основі дрібних помилок оцінити загальне число помилок, що залишилися в програмі.

Підхід базується на таких поняттях:

* **мутації** – дрібні помилки в програмі;
* **мутанти** – програми, що відрізняються одна від одної мутаціями.

Метод мутаційного тестування полягає в тому, що в програми А та Б, що розробляються, вносять мутації, тобто штучно створюють програми-мутанти А1, Б1... Потім програми А та Б і їх мутанти тестуються на одному і тому ж наборі тестів (X,Y).

Якщо на наборі (X,Y) підтверджується правильність програми P і, крім того, виявляються всі внесені до програм-мутантів помилки, то набір тестів (X,Y) відповідає мутаційному критерію, а тестована програма оголошується правильною.

Якщо деякі мутанти не виявили всіх мутацій, то треба розширювати набір тестів (X,Y) і продовжувати тестування.

***Для самостійного вивчення***: Поглибити матеріал лекції за наданою літературою. Вивчення лекційного матеріалу та додаткових джерел. Розгляд запитань і виконання завдань для самостійної роботи, запропонованих на лекції.

***Література***

1. Лавріщева К.М. Програмна інженерія. Електронний підручник: – URL: <http://csc.knu.ua/uk/library/books/lavrishcheva-6.pdf>
2. Сайкс Д. Тестирование объектно-ориентированного программного обеспечения. Практическое пособие / Д.Сайкс, Д.Макгрегор. –К.: Диасофт, 2002. –432 с.
3. Соммервил И. Инженерия программного обеспечения / И.Соммервил.–М. : Издательский дом «Вильямс», 2002. –623 с.
4. Тамре Л. Введение в тестирование программного обеспечения / Л. Тамре. –М.: Издательский дом «Вильямс», 2003. –368 с.
5. Блэк Р. Ключевые процессы тестирования / Р.Блэк. – М.: Лори, 2006. –544 с.
6. Винниченко И.В. Автоматизация процессов тестирования / И.В.Винниченко. – СПб.: Питер, 2005. –208 с.
7. Криспин Л. Гибкое тестирование. Практическое руководство для тестировщиков ПО и гибких команд / Л.Криспин, Д.Грегори. –М.: Издательский дом «Вильямс», 2010. –464 с.
8. Майерс Г. Искусство тестирования программ / Г. Майерс. — Пер. с англ.— М.: Финансы и статистика, 1982. — 172 с.
9. Фолк Д. Тестирование программного обеспечения / Д.Фолк, Е.К. Нгуен, С.Канер. – К.: Диасофт, 2003. –400 с.

**Контрольні запитання.**

1. Які типи тестових прикладів існують?
2. Назвіть основні вимоги до ідеального критерію тестування.
3. Особливості застосування структурних і функціональних критеріїв.
4. Яка різниця між тестуванням гілок і тестуванням шляхів?
5. Що собою представляє метод оцінки швидкості виявлення помилок?
6. Що таке мутаційний критерій і в чому полягає техніка роботи з ним?